
Universidade do Estado do Rio de Janeiro
Instituto de Física

Phys-Pub 027/95
Preprint
October 1995

Symbolic Computing with Grassman Variables

E.S. Cheb-Terrab¹

Abstract

A package of Maple 5.3 commands for doing calculations with anticommutative variables is presented.

(Submitted to Journal of Symbolic Computation)

¹Departamento de Física Teórica, IF-UERJ. E-mail: terrab@vmesa.uerj.br.

1 Introduction

Working with anticommutative symbolic objects is becoming part of the standard activity in mathematical-physics (Sohnius, 1985). On the other hand, almost all available general-purpose symbolic computing systems have not yet included anticommutative variables (ACVs) in their default computational domains. In Maple, for instance, this is reflected in the product operator, `*`, defined as commutative, and in the standard differentiation command, `diff`, not designed to evaluate derivatives with w.r.t ACVs. Also, the admitted non-commutative product operator, `&*`, does not have its differentiation rule and its basic simplification operations defined.

This paper presents a set of Maple 5.3 commands for working with ACVs, covering the topics mentioned above. The exposition is organized as follows. In Sec.2, a brief review of the package is presented. In Sec.3, the extension of the computational domain and the rules for manipulating and simplifying anticommutative products are discussed. Sec.4 is dedicated to the differentiation problem; that is, the rules for differentiating anticommutative products, composite functions, and for obtaining high order derivatives w.r.t. ACVs. Sec.5 contains a brief illustration of how the new commands work. Finally, the Conclusions contain some general remarks about this work and its possible extensions.

2 Anti-commutative variables and the *Grassman* package

To start with, let's recall that any pair $\{\theta_i, \theta_j\}$, belonging to a set $\{\theta_1, \dots, \theta_n\}$ of *anticommutative* variables, satisfies² $\theta_i \& * \theta_j = -\theta_j \& * \theta_i$. As a convention, the **parity** is here defined as 1 for all θ_n variables, and 0 for any other (*commutative*) variable. These definitions are extended to include the concepts of *commutative*, $F(\dots)$, and *anticommutative*, $Q(\dots)$, functions, with parity defined as 0 and 1, respectively. Note that both $F(\dots)$ and $Q(\dots)$ may or not depend on the anticommutative variables θ_n .

The package of commands here presented allows its user to work with *anticommutative* variables and functions by introducing algebraic symbols to represent them, and by defining their basic operations and differentiation rules taking into account their *anticommutative* character. A brief review of the package is as follows:

- all variables built with the string `'theta'` followed by a positive number are considered *anticommutative* variables; all functions with name beginning with the capital letter Q are considered *anticommutative* functions;
- three type subroutines, `'type/grassman'`, `'type/commutative'` and `'type/anticommutative'`, and a command, **parity**, permit the recognition of the *commutative* and *anticommutative* character of any algebraic symbol or expression;
- the **midentity** command replaces nested, not simplified, *non-commutative* `&*` products, in a given expression, by "un-nested" and simplified ones;

²Following Maple conventions, the symbol used by the package's commands to represent a *non-commutative* product operator is `&*`.

- the **msort** command sorts the operands of each $\&*$ product of a given expression in a unique manner, permitting an efficient manipulation of the expression by the Maple system as a whole;
- the **mexpand** command sorts (using **msort**) and recursively expands the $\&*$ products over sums in a given expression;
- the **gdiff** command is a differentiation command designed to work both with commutative and anticommutative variables.
- the set of commands **useD**, **usegdiff** and **usediff** change the format of the derivatives of a given expression to the **D**, **gdiff** and **diff** format, respectively;
- the utility subroutines '**diff/&***' and '**diff/function**' define the rules for differentiating $\&*$ products and composite functions, taking into account the possible *anticommutative* character of the symbols involved.

Of the eight commands of the package, **mexpand** and **gdiff** are the key ones. The former makes calls to **msort**, which in turn makes calls to **midentity**, producing a somewhat elaborated result, while **gdiff** is a generalization of the standard **diff** command; working as the latter concerning syntax and display and when the differentiation variable is not an ACV.

3 The computational domain and the *non-commutative* $\&*$ products

3.1 The '**type/...**' subroutines

As mentioned in the Introduction, the default computational domain of the Maple system does not include anticommutative variables. Hence, the departure point was the development of three subroutines, '**type/grassman**', '**type/commutative**', '**type/anticommutative**', and a command, **parity**, to recognize the possible commutative/anticommutative character of a given expression. The '**type/grassman**' subroutine identifies an anticommutative variable or function, as follows³:

Calling sequence:

```
> type(var,grassman);
```

Scheme for the answers:

- if **var** is a string *and* the first five characters are **theta** *and* the remaining part of the string is a positive integer *then* **true**
- if (**var** is a string *and* the first character is **Q**) *or* ((**var** is a function *or* an indexed object) *and* the first character of its name⁴ is **Q**) *then* **true**

³In what follows, the *input* can be recognized by the Maple prompt **>**.

⁴In Maple, the name of such objects is given by **> op(0,var);**

- *else* **false**

The **'type/commutative'** subroutine addresses a more general task, which is to determine whether a whole algebraic expression, arbitrarily composed with sums, products, powers, functions and constants, *is or is not* commutative; it relies completely on the **'type/grassman'** subroutine, and works as follows:

Calling sequence:

```
> type(expr,commutative);
```

Scheme for the answers:

- *if* **expr** is a sum *then* the subroutine makes a recursive calling of itself *and if* each operand is commutative *then* **true**
- *if* **expr** is a product *then* the subroutine makes a recursive calling of itself *and if* there is an even number of non-commutative operands *then* **true**
- *if* **expr** is a power *then if* the base is commutative or the exponent is even *then* **true**
- *else* send the task to **'type/grassman'**, returning **true** or **false** correspondingly.

Note that an expression is being recognized as *commutative* just in the “positive” case, that is, when this commutativity is actually verified. For instance, the **'type/anticommutative'** subroutine works in almost the opposite manner, but an expression can be neither *commutative* nor *anticommutative*; for example, a sum of objects of each type.

Finally, the **parity** command is more like a “macro”, and works as follows:

Calling sequence:

```
> parity(expr);
```

Scheme for the answers:

- *if* **expr** is of commutative type *then* 0
- *if* **expr** is of anticommutative type *then* 1
- *else* return the string *'undefined'*.

The three **'type/...'** subroutines and the **parity** command, are recursively used by the other commands of the package before the decision of what to do with the received arguments is taken. In this manner, though it is not possible to introduce changes in Maple's kernel, it is possible to introduce the tools for making correct evaluations of the commutative/anticommutative character of an algebraic symbol. Then, the correct results can be obtained by using the appropriate commands, mainly **&*** for the product of objects containing ACVs, **mexpand** for simplifying these products and **gdif** for evaluating derivatives.

3.2 The *non-commutative* $\&*$

To represent non-commutative products, the Maple system uses the $\&*$ operator; it has the correct precedence with regard to the other basic operations, but no elementary simplification or differentiation rules for it are defined. Therefore, the idea was to take advantage of the precedence rules of $\&*$ and make it work as a *commutative* or *anticommutative* product, according to the parity of its operands.

This is accomplished by the package's commands **midentity**, **msort**, **mexpand** and **gdif**, when they are called, by substituting all occurrences of $\&*$ by a subroutine $\&p$, which actually realizes all the calculations. The idea was not to assign anything to the $\&*$ operator, in order to maintain the compatibility with the other commands of the Maple Standard Library (MSL) which make calls to it.

Additionally, many usually desired simplifications of this $\&*$ product are automatically accomplished by the $\&p$ routine. These simplifications are at the base of all the subsequent calculations, and can be summarized as follows:

Calling sequence:

```
> &p(a,b,c,...);
```

Scheme for the answers:

- if there is only one operand, say a , then a
- if 0 belongs to the sequence of operands then 0
- if an operand, say a , is in turn a product such as $\&*(f,g,h)$ or fgh , then replace it by its operands f,g,h and *evaluate the whole expression again*;
- else return an expression of the form $\alpha \&*(...)$, where α is build as a standard product of all the operands of $\&p(a,b,c,...)$ satisfying the condition of *not having grassman-type variables or functions*.

Note that the third item is applied recursively and, as a consequence, an arbitrarily nested structure containing combinations of $\&*$ and $'*'$ will always be reduced to a plain structure, with all the operands inside the $\&*$ product on the same level, and with all the operands not having grassman-type objects as factors outside. For example, the process above will map nested products as in

$$\&*(a \&*(F(\theta), b \&*(Q_1(\theta), c Q_2(\theta)))) \rightarrow a b c \&*(F(\theta), Q_1(\theta), Q_2(\theta))$$

The $\&p$ operator may keep objects of parity = 0 inside a $\&*$ product, even when this objects actually commute with all the other ones, for instance $F(\theta)$ in the example above. The reason for this is related to the differentiation w.r.t θ_n variables (objects of parity one), and can be seen by considering the product of a parity = 0 function, say $F(\theta)$, times a parity = 1 function, say $Q(\theta)$. This product satisfies:

$$F(\theta) Q(\theta) = Q(\theta) F(\theta)$$

Differentiating the left-hand-side (lhs), for example, one receives

$$\frac{\partial F(\theta)}{\partial \theta} Q(\theta) + F(\theta) \frac{\partial Q(\theta)}{\partial \theta}$$

Since the operands of $\frac{\partial F(\theta)}{\partial \theta} Q(\theta)$ does not commute (both now have parity = 1), this result is correct *if one maintains the order of the operands*, that is, *if* the product is represented using the *non-commutative* operator $\&*$. On the other hand, if one begins using the standard ‘ $*$ ’ for representing the *commutative* product FQ , then one should expect the result to be expressed in terms of the *commutative* ‘ $*$ ’ too, which in turn will be wrong.

Thus, in the context of the package here presented, to obtain the correct results, the $\&*$ operator should be used to represent both commutative and non-commutative products, while $\&p$ will take care of keeping as arguments just the operands having grassman-type objects inside. The surface command for achieving these simplifications is **midentity**.

A “canonical form” for $\&*$ (a, b, c, \dots)

In order to make more efficient the simplification of expressions containing $\&*$ products by the commands of the MSL, a “canonical form” for these products was implemented; that is, a “unique” manner of writing equivalent but apparently different such products. The command accomplishing this process is **msort**, which makes calls to **midentity** and works as follows:

Calling sequence:

```
> msort(expr);
```

Scheme for the answer: All $\&*$ products of **expr** are sorted according to:

- the sequence of arguments of $\&*$ is split in two sequences, s_1 and s_2 , containing all the *commutative* objects, and all the *non-commutative* ones, respectively;
- the elements of s_1 are sorted using machine ordering (**table(symmetric)**)
- the elements of s_2 are sorted using machine ordering (**table(antisymmetric)**), and a -1 factor is introduced when the number of required permutations is odd.

Also, this mechanism automatically evaluates to 0 all $\&*$ products containing powers of non-commutative symbols.

The **mexpand** command addresses the problem of distributing $\&*$ products over sums, makes calls to **msort**, and also works recursively. This distribution of $\&*$ products over sums will be relevant both in the process of sorting the $\&*$ products explained above and in taking advantage of the set of simplification commands of the MSL.

4 Differentiation of expressions containing *grassman-type* symbols

The introduction of anticommutative objects in the computational domain required both the extension of the differentiation rules and the definition of equivalencies between the standard **D** and a new **gdiff** differentiation operators.

4.1 Differentiation of $\&*$ products, composite functions and high order derivatives

The differentiation operation was extended by defining the differentiation rules for $\&*$ products and composite functions, and high order derivatives. This was accomplished by building a ‘**diff**/ $\&*$ ’ subroutine, adapting the standard ‘**diff/function**’, and creating a new differentiation command, **gdiff**, which in turn relies completely on the standard **diff**.

The differentiation rule for products, ‘**diff**/ $\&*$ ’, was programmed to return a result according to

$$\frac{\partial}{\partial \theta}(A \&* B) = \frac{\partial A}{\partial \theta} \&* B + (-1)^{\text{parity}(A)} A \&* \frac{\partial B}{\partial \theta}$$

The differentiation rule for composite functions was programmed introducing a few lines in the standard ‘**diff/function**’, taking care *not to change anything* in its standard behavior with regard to commutative objects, but returning results according to:

$$\frac{\partial}{\partial \theta} \left(A(B) \right) = \frac{\partial B}{\partial \theta} \&* \frac{\partial A}{\partial B}$$

when grassman-type objects are involved in the “derivand”.

To request the evaluation of a derivative taking into account the *non-commutative* character of the involved symbols one should use the **gdiff** command. The syntax of **gdiff** is that of the standard **diff**, and a ‘**print/gdiff**’ procedure was created in order to obtain the same display too. A brief summary of how this command works is as follows.

Calling sequence:

```
> gdiff(expr,x);
```

Scheme for the answers:

- if (**x** is not of grassman-type) *or* (there are no derivatives within **expr**) *then* send the task to the standard **diff**
- if **expr** is a derivative *then*
 1. split the sequence of differentiation variables into two sequences, s_c and s_{nc} , containing the commutative and non-commutative variables, respectively;
 2. differentiate the derivand w.r.t the commutative variables s_c using **diff**;
 3. sort the s_{nc} elements using machine ordering, differentiate the result of the previous step w.r.t the sorted s_{nc} grassman variables, using **gdiff**, and introduce a -1 factor when an odd number of permutations was required⁵.
- if **expr** contains, say n derivatives *then*
 1. substitute the n^{th} derivative by a function $Y_n(x)$, where Y is a local variable;
 2. differentiate the resulting expression w.r.t x , using **diff**, obtaining an expression involving the symbols $\frac{\partial}{\partial x} Y_n(x)$;

⁵To avoid infinite recursion, the result of this step is returned “built”, but “unevaluated”.

3. differentiate the n^{th} derivative found in **expr** w.r.t x , using **gdiff**, and build a set of equivalencies $\{\frac{\partial}{\partial x}Y_n(x) = R_n\}$, where R_n represents the obtained results;
4. introduce these equivalencies in the result of step 2, sort all $\&*$ products using **msort** and return a result

4.2 Equivalencies between the **D** and **gdiff** differentiation operators

The Maple system includes two differentiation formats for representing derivatives, related to the **diff** and the **D** operators, respectively. Although the first one is more intuitive, the second one is more general: *all* derivatives can be represented using the **D** operator while *just some* using **diff** (Monagan and Devitt, 1992). The conversion between formats in a given algebraic expression can be realized by means of the **convert** command.

Having the option of representing *all* derivatives in a unique manner, as is the case of the **D** format, is a highly desirable property of a symbolic computing system. Hence, the idea was to define a correspondence between the new **gdiff** and the **D** formats too. With this purpose in mind, consider, for instance, the standard representations for a second order derivative:

$$\frac{\partial^2}{\partial \theta_1 \partial \theta_2} f(\theta_1, \theta_2, \theta_3) = D_{1,2}(f)(\theta_1, \theta_2, \theta_3) \quad (1)$$

Since high order derivatives are supposed to commute, the order in which θ_1 and θ_2 appear in the lhs, related to the **diff** format, is irrelevant and “session dependent” (machine ordering). On the other hand, the **D** representation in the rhs carries an “additional information”: the ordering of the numbers 1 and 2, which will always be numerical order. This property of the **D** format can be used to address the fact that, when differentiating w.r.t ACVs, the order in which the derivatives are realized must be taken into account. Therefore, the equivalence between the **gdiff** and **D** formats was proposed as it appears in Eq. (1), now admitting that the lhs was built using **gdiff**, thus preserving both the order in which θ_1 and θ_2 are displayed and that in which the derivatives are realized.

The conversion between equivalent formats was implemented by adapting the **useD** and **usediff** commands of the *partials* package (Cheb-Terrab, 1994) and building a new one, **usegdiff**. The “names” these commands have resemble the “actions” they realize, and the ‘**convert/...**’ facility was not used in order to keep as much as possible the compatibility with the other commands of the MSL⁶. Note also that, while it was possible to state conversion rules between the **gdiff** and **D** formats, this conversion is *not possible* between **gdiff** and **diff**, since high order derivatives represented using the latter are assumed by the whole system to commute.

5 Examples

This section aims at giving a brief illustration of how the new commands work. To start with, let’s load the code and introduce some macros and alias that improve the readability

⁶The subroutines ‘**convert/D**’ and ‘**convert/diff**’ already exist and work a bit differently from **useD** and **usediff**.

of the *input/output*⁷ of θ_1, θ_2 , the functions $Q_1(x, y, \theta_1, \theta_2)$, $f_1(x, y, \theta_1, \theta_2)$, $f_2(x, y, \theta_1, \theta_2)$ and the composite function $Q_2(\theta_1, \theta_2, Q_1)$

```
> read grassman;
> t1=theta1, t2=theta2;
> map(macro, [""]):
```

$$t1 = \theta_1, t2 = \theta_2$$

```
> (Q[1]=Q[1](x,y,theta1,theta2), Q[2]=...
> map(alias, [""]):
```

$$Q_1 = Q_1(x, y, \theta_1, \theta_2), Q_2 = Q_2(\theta_1, \theta_2, Q_1), f_1 = f_1(x, y, \theta_1, \theta_2), f_2 = f_2(x, y) \quad (2)$$

The ‘**type/grassman**’ subroutine recognizes *non-commutative* objects as in:

```
> [t1, t2, Q[1], Q[2], f[1], f[2]]:
> map(type, "", grassman);
```

$$[\text{true}, \text{true}, \text{true}, \text{true}, \text{false}, \text{false}] \quad (3)$$

The **parity** command can be used to determine the *parity* of composite expressions:

```
> [Q[1]+Q[2], f[1]+f[2], Q[1]+F[2]]:
> map(parity, "");
```

$$[1, 0, \text{undefined}] \quad (4)$$

Consider the *commutative* $\&*$ product of a parity = 1 object times a parity = 0 one:

```
> Q[1] &* (f[1] + f[2]);
```

$$Q_1 \&* (f_1 + f_2) \quad (5)$$

The **msort** command sorts this product taking into account its commutativity:

```
> msort("");
```

$$(f_1 + f_2) \&* Q_1 \quad (6)$$

and only objects *having grassman-type variables* are kept inside a $\&*$ product:

```
> mexpand("");
```

$$(f_1 \&* Q_1) + f_2 Q_1 \quad (7)$$

Consider now a nested $\&*$ product:

```
> ((a*Q[1] &* (b*Q[2])) &* (Q[2]+f[1]+f[2])) &* f[2];
```

$$((a Q_1 \&* b Q_2) \&* (Q_2 + f_1 + f_2)) \&* f_2 \quad (8)$$

The **midentity** command transform it into an “un-nested” $\&*$ product:

```
> midentity("");
```

$$a b f_2 \&* (Q_1, Q_2, Q_2 + f_1 + f_2) \quad (9)$$

The **mexpand** command automatically cancels products such as $Q_2 \&* Q_2$:

```
> mexpand("");
```

$$a b f_2 (\&* (f_1, Q_1, Q_2) + f_2 (Q_1 \&* Q_2)) \quad (10)$$

⁷In Maple, “” means the last expression and “”” means the second last expression. Also, special care was taken to keep the *input* and *output* shown along this paper with almost exactly the same format and aspect as that which appears on the computer screen.

Differentiation examples

To start with, let's consider the mixed derivatives of the grassman function Q_1 , w.r.t the grassman variables θ_1 and θ_2 , and check the anticommutativity of these derivatives:

```
> q[t1,t2] := gdiff(Q[1],t1,t2);
```

$$q_{\theta_1,\theta_2} := \frac{\partial^2}{\partial \theta_2 \partial \theta_1} Q_1 \quad (11)$$

```
> q[t2,t1] := gdiff(Q[1],t2,t1);
```

$$q_{\theta_2,\theta_1} := -\frac{\partial^2}{\partial \theta_2 \partial \theta_1} Q_1 \quad (12)$$

The conversion between the **D** and the **gdiff** formats is accomplished via:

```
> useD(q[t1,t2]) = usegdiff(useD(q[t1,t2]));
```

$$D_{3,4}(Q_1)(x, y, \theta_1, \theta_2) = \frac{\partial^2}{\partial \theta_2 \partial \theta_1} Q_1 \quad (13)$$

Derivatives w.r.t *commutative* variables are displayed separately:

```
> gdiff(Q[1],t2,x,t1,y);
```

$$-\frac{\partial^2}{\partial \theta_2 \partial \theta_1} \left(\frac{\partial^2}{\partial y \partial x} Q_1 \right) \quad (14)$$

Finally, consider the second order derivatives of the $\&*$ product of Q_1 with the composite function $Q_2(\theta_1, \theta_2, Q_1)$:

```
> q[0] := Q[1] &* Q[2]:
```

```
> gdiff("",t1);
```

$$\begin{aligned} & \left(\left(\frac{\partial}{\partial \theta_1} Q_1 \right) \&* Q_2 \right) \\ & - \left(\left(D_1(Q_2)(\theta_1, \theta_2, Q_1) + \left(D_3(Q_2)(\theta_1, \theta_2, Q_1) \&* \left(\frac{\partial}{\partial \theta_1} Q_1 \right) \right) \right) \&* Q_1 \right) \end{aligned} \quad (15)$$

```
> q[t1,t1] := mexpand(gdiff("",t1));
```

$$q_{\theta_1,\theta_1} := 0 \quad (16)$$

```
> gdiff(q[0],t1,t2) + gdiff(q[0],t2,t1);
```

$$0 \quad (17)$$

6 Conclusions

The main idea of this paper is the possible extension of the computational domain of general-purpose symbolic systems to include ACVs. This extension was here presented, for the Maple system, as a set of type subroutines and commands for evaluating products and derivatives taking into account the anti-commutative character of the symbols involved.

Concerning the compatibility of the new routines with the system as a whole, it is good, since almost no any assignments were made to standard routines, except for the ‘**diff/function**’ subroutine. On the other hand, some commands of the MSL may attempt to convert **D** derivatives to the **diff** format while making calculations, thus introducing errors when high order derivatives w.r.t anti-commutative variables are involved.

To conclude, this work can be extended by introducing symbols to represent indexed grassman functions associated with spinor representations of space-time, for instance, including the rules for manipulating the γ^μ Dirac matrices. Such an implementation would be relevant in the introduction of general-purpose symbolic computing systems in theoretical physics research.

Acknowledgments

This work was supported by the State University of Rio de Janeiro (UERJ) and the National Research Council (CNPq) of Brazil. The author would like to thank Ali Ayari, from the Centre de recherches mathématiques, Université de Montréal for useful discussions.

References

- [1] Sohnius, M. F. (1985). Introducing Supersymmetry. *Physics Reports* **128**, Nos. 2 & 3, 39–204.
- [2] Monagan, M., Devitt, J.S. (1992). The D Operator and Algorithmic Differentiation. *The Maple Technical Newsletter* **7**, 17–24.
- [3] Cheb-Terrab, E.S. (1994). Maple procedures for partial and functional derivatives. *Computer Physics Communications*, **79**, 409–424.